Optimization of a Wordle algorithm[1]

Morten Heine Sørensen
mhs@formalit.dk

When you have eliminated all which is impossible,
then whatever remains,
however improbable,
must be the truth.

Sherlock Holmes
In *Arthur Conan Doyle, The Case Book of Sherlock Holmes.*

ABSTRACT
We present a Wordle guessing algorithm implemented in Python and show, in a number of steps, how it can be heavily optimized, reducing running time for all secrets from several hundred days down to 2 minutes. In one of the steps, the code is rewritten in C++. The algorithm requires at most 7 guesses for any secret. By manual handling of some cases, it is known that at most 6 guesses are required.

INTRODUCTION
The game of *Wordle* has recently become popular. In 6 attempts the player must guess a secret English word of 5 letters. After each attempt, the player receives feedback about the guess; each correct letter in a correct position is marked green, and each correct letter in wrong position is marked yellow.

The game is related to *Mastermind*, where the player must guess an ordered combination of 4 pegs, where the pegs come in 6 different colors. After each attempt, the player receives feedback about the guess; a correct color in correct position gives a black stick, and a correct color in wrong position gives a white stick.

It is well known that an effective strategy, due to Knuth, exists that always produces the correct guess in at most 5 guesses [6].

A variant, known as *Super Mastermind*, works instead with 5 pegs each of 8 different colors, which makes the Wordle analogy even closer. Again, it is well-known that an effective strategy exists that produces the correct guess in at most 8 guesses [7].

The question arises whether an effective strategy exists, relative to some list of English words, that can guess the secret word in at most the 6 available attempts.

In this paper we present a version of Knuth's algorithm that can guess any Wordle secret in at most 7 attempts. The algorithm was implemented and heavily optimized. Initially it would run for several hundred days to complete all games, in the end it was reduced to around 15 minutes.

During preparation of the paper, it was reported that, indeed, Knuth's algorithm, with some manual tinkering, can be used to guess any word in at most 6 attempts [8].

The most interesting aspects of the strategy are:

- It does not try to identify the correct guess, but rather attempt to eliminate as many wrong guesses as possible.
- It deliberately makes guesses, which it knows cannot be the correct word.

WORDLE AND SUPER MASTERMIND
The game of Wordle was developed by Josh Wardle and made public in October 2021 [1]. The author became aware of it in Denmark sometime early January.

It is an online game, where the player is to guess a secret 5 letter word in at most 6 attempts, see Figure 1.

Figure 1. The game of Wordle

[1] This is a revised version 4 of the paper. I am indebted to my co-workers in Sprinting Software for helping me run the full set of games on an early version of the code: Aleksa Ateljevic, Boris Demir, Djordjije Matic, Jasmina Staniukovic, Jonatan Kutchinsky, Vasil Manolov and Zlabitor Veljkovic. Ivan Olkhovskii wrote most of the C++ code and Jonatan Kutchinsky improved the Python code to use numPy.

The guess must in each case be a correct word, otherwise it is rejected, and not counted as an attempt. So, there is a list of words from which the secrets are drawn and which the guesses are validated against. It has been revealed from the source code that the list is the Collins official Scrabble Words 2019 list [2], limited to the 12972 words consisting of 5 letters.

After each attempt, the player receives feedback about the guess:

- Each correct letter in correct position is marked green.
- Each correct letter in wrong position is marked yellow.

Notice that it is apparent from the feedback not only that there is *some* correct letter in a correct (or wrong) position, but which one it is. In the second row, for instance, we see that the "N" is a correct letter in a correct position. Similarly, we see that the "O" is a correct letter in wrong position.

As mentioned, the game of Super Mastermind is similar. It is a two-player game, where one player devises a secret combination of colored pegs, which the other player has to guess, see Figure 2. The secret consists of 5 pegs, each of which have one among 8 colors. Multiple pegs may have the same color.

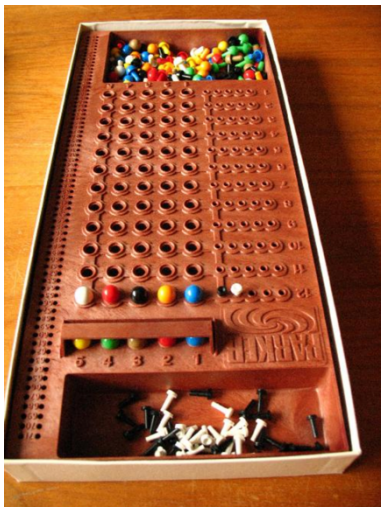

Figure 2. Super Mastermind

We do not go into details concerning how the players compete, but simply compare a single round of Super Mastermind with a single round of Wordle, in both cases taking the point of view of the player who has to guess the secret.

In Super Mastermind, the player guessing the secret has a total of 12 attempts. For each attempt, the player receives feedback about the guess:

- Each correct color in correct position gives a black stick.
- Each correct color in wrong position gives a white stick.

Notice that the order of the black and white sticks is not significant. In other words, when the player receives, e.g,. a black stick, he does not know which of the pegs have the correct color and position, only that one of the pegs have the correct color and position.

Another point worth emphasizing is that each peg in the guess can score against only one peg in the secret, e.g.:

- If the secret has one green peg and the guess has two green pegs, one of which has the correct position, the score will be a black stick, not a black and white stick.
- If the secret has two yellow pegs and the guess has one, which is not placed in the position of any of the yellow pegs in the secret, the result is one white stick, not two.
- If the secret has one yellow peg and the guess has two, none of which is placed in the position of the yellow peg in the secret, the result is one white stick, not two.

This principle applies also to Wordle. For instance, if the secret has one "O" and the guess has two, none of which is placed correctly, only one of them will be marked Yellow. We assume it is the leftmost of the two.

In summary, the similarities and differences of Wordle and Super Mastermind are as indicated in Table 1.

Principle	MM	Super MM	Wordle
Holes	4	5	5
Peg colors	6	8	26
Valid combinations	All	All	From list
Position of peg with right color and right/wrong position	Not known	Not known	Known
Number of different scores	15	17	243
Number of different secrets	1296	32.768	12.972
Maximum guesses available	8	12	6
Maximum guess needed	5	7	6

Table 1. Super Mastermind vs Wordle

It may seem that Wordle is far more complex than Super Master-mind due to the larger number of combinations:

- For Super Mastermind: $8^5 = 32.768$
- For Wordle: $28^5 = 17.210.368$

However, there is a smaller number of 5-word letters in the list, namely 12972. In this sense, Wordle is simpler than Super Mastermind.

On the other hand, the number of possible scores of a guess, which is an important ingredient in the winning strategy in both games, is greater in Wordle.

SUPER MASTERMIND WINNING STRATEGY

We now present the algorithm for Super Mastermind which always produces the correct answer in at most 8 attempts. But first some notation.

By score(C,S) denote the score of the guess C against the secret S. Concrete scores will be denoted B*n*W*m*, meaning *n* black sticks and *m* white sticks. Let A denote the set of all combinations Let s_W be the winning score B5W0.

Algorithm 1. Super Mastermind.

1. Let S be the secret.
2. Let C be the first guess from A.
3. Let s = score(C,S).
4. If s= s_W, C is the secret.
5. Otherwise, proceed as follows.
6. Remove from A every C' with score(C',S)≠s.
7. Pick a new guess C'' and go to Step 3 with C=C''

It remains to explain two parts in detail:

- The choice of initial guess in step 2.
- The picking of a new guess in step 7.

Algorithm 2. Next Guess in Super Mastermind.

1. For each remaining combination C in A:
   a. For each score s of C against any combination in A, let m(C,s) be the number of combinations C' in A with score(C,C')=s.
   b. Let M(C) be the **maximal** m(C,s) for any s.
2. Let M be the **minimal** M(C) for any C. We call this the *minmax* value of C.

In words, for each combination, we first calculate the maximum number of combinations that can remain after guessing C, for any score, and then select the combination where that maximum is as small as possible. The selected combination is the one that has the best guaranteed reduction of the set of remaining possibilities.

Algorithm 3. Initial guess in Super Mastermind.
For the initial guess, we can simply run the next guess algorithm on the initial set.

It turns out that this algorithm is not optimal. It is possible to improve the algorithm so that it has a lower upper bound on the number of guesses it will make. The improvement is as follows.

Algorithm 4. Improved next guess in Super Mastermind.
In step 1 pick the new guess C from the initial set A of all combinations rather than the set of remaining combinations. If there are both guesses outside and inside the reduced set with the same number N, prefer one in the reduced set. When multiple guesses have the same number, choose the first one in alphabetical order.

The reason why this algorithm requires fewer guesses is that it can more effectively eliminate remaining combinations using the best suited words in the initial set, even if they are no longer in the reduced set.

WORDLE WINNING STRATEGY

Next, we turn to Wordle. We view this as a Super Mastermind game the following way:

- Each letter is considered a color.
- Each position of a letter is considered a hole.

The difference is that the scores are slightly different. We will denote a score by a sequence of 5 trinary digits, where

- 2 means correct letter in correct position
- 1 means correct letter in incorrect position
- 0 means incorrect letter.

For instance, with secret PANIC, the guess PHONY gets score 20010.

The basic algorithm is exactly that same as Algorithm 1-4. The initial list A is that of all actual 5 letter words. The start word becomes SERAI. With this one we get maximum number of guesses 8, see Table 2.

Guesses	Count	Avg
1	1	1
2	66	132
3	1740	5220
4	6412	25648
5	4059	20295
6	652	3912
7	38	266
8	4	32
SUM	**12972**	**4,27890842**

Table 2. Wordle starting with SERAI

Strictly speaking, this table is the result of running with a slight variation of Algorithm 2. We return to this in the next section.

It is well known that the start word matters and that different next guesses with same minmax value may lead to different number of guesses. In other words, it makes sense to play around with the initial guess and the choice of next guess among multiple with the same or similar minmax value.

We may find inspiration in the frequency of each letter in each position in the known list of words:

LETTER	HOLE 1	HOLE 2	HOLE 3	HOLE 4	HOLE 5
A	0,06	**0,17**	**0,10**	0,08	0,05
B	0,07	0,01	0,03	0,02	0,00
C	**0,07**	0,01	0,03	0,03	0,01
D	0,05	0,01	0,03	0,04	0,06
E	0,02	0,13	0,07	**0,18**	0,12
F	0,05	0,00	0,01	0,02	0,01
G	0,05	0,01	0,03	0,03	0,01
H	0,04	0,04	0,01	0,02	0,03
I	0,01	0,11	0,08	0,07	0,02
J	0,02	0,00	0,00	0,00	0,00
K	0,03	0,01	0,02	0,04	0,02
L	0,04	0,05	0,07	0,06	0,04
M	0,05	0,01	0,04	0,03	0,01
N	0,03	0,03	0,07	0,06	0,04
O	0,02	0,16	0,08	0,05	0,03
P	0,07	0,02	0,03	0,03	0,01
Q	0,01	0,00	0,00	0,00	0,00
R	0,05	0,07	**0,09**	0,06	0,05
S	**0,12**	0,01	0,04	0,04	**0,31**
T	0,06	0,02	0,05	0,07	0,06
U	0,01	0,09	0,05	0,03	0,01
V	0,02	0,00	0,02	0,01	0,00
W	0,03	0,01	0,02	0,01	0,00
X	0,00	0,00	0,01	0,00	0,01
Y	0,01	0,02	0,02	0,01	0,10
Z	0,01	0,00	0,01	0,01	0,00
MAX	**0,12**	**0,17**	**0,10**	**0,18**	**0,31**
	C	A	R	E	S
	4	**3**	**5**	**2**	**1**

Table 3. Letter frequencies

Assuming that we do not want the same letter multiple times, we can choose CARES as initial word of maximal frequency this way. This leads to maximally 8 guesses for just one word.

If we make further experiments with the letter with the lowest frequency (the first letter), we can choose them one by one and with LARES it turns out that we need only 7 guesses, at least when we use another slight variation of the algorithm, as explain in the next sections.

OPTIMIZATIONS

The runs mentioned in the preceding section were done with a program, see Table 4 ("~" indicates *estimated* time).

Code version	Lang	Time 1 game	Time total	Mem	*
Initial version	Python	> 85m	> 779d	< 0,5 Gb	-
MinMax	Python	> 25m	> 223d	< 0,5 Gb	3.5
Precomp scores	Python	1m 2m preload	~9d	11,5 Gb	25
numPy	Python	1m30s 2m preload	~13.5d	3,5 Gb	-
numPy 3 Thread	Python	30s 2m preload	~4.5d	10,5 Gb	2
Real scores	Python	5s 2m preload	~18h	3,5 Gb	(18)
Real scores 3 Thread	Python	1,7s 2m preload	~6h	10,5 Gb	18
Cache guesses 3 Thread	Python	0,28s	~1h	10,5 Gb	6
C++ All optim except guesses 1 thread	C++	0,5s 2m preload	~110m 2m preload	0,5 Gb	(9.8)
C++ All optim except guesses 16 thread	C++	~0,07s 2m preload	15m 2m preload	0,5 Gb	24
C++ All optim 16 thread	C++	~ 0,018s	2.5m 4m preload	8 Gb	7,5

Table 4. Wordle starting with LARES

The initial program was a Python program written generically to handle any Mastermind case with an arbitrary number of holes and colors as well as handling the Wordle case for any wordlist, alphabet, and number of letters. Mainly testing scores and checking scores for equality were different between the game variants.

Initial Version

Due to the generic form of the code and pending performance measurements, the initial version was slow. The initial version would run for around 2 years to check the number of guesses required for all secrets.

MinMax Without Superfluous Calculation

The first optimization was to slightly improve the calculation of minmax in the next guess algorithm.

The calculation of the minmax value for each possible guess was changed so that the minimum calculated so far, for the previous guesses, was passed to the calculation of the next guess' maximum. As soon as the calculation of the latter maximum exceeds the minimum computed so far, the calculation of that maximum can be abandoned, as it will not lead to a new minimum.

This resulted in a reduction of run time with factor 3.5

Precomputing the Scores
Running a profiling session with the revised code, revealed that most time was spent calculating scores, see Figure 3.

```
         1884630907 function calls (1884630541 primitive calls) in 710.813 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  710.813  710.813 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 _bootlocale.py:33(getpreferredencoding)
        1    0.000    0.000    0.000    0.000 _weakrefset.py:38(_remove)
        1    0.000    0.000    0.000    0.000 codecs.py:186(__init__)
        2    0.000    0.000    0.000    0.000 codecs.py:260(__init__)
    12972    0.001    0.000    0.001    0.000 codecs.py:276(reset)
        2    0.000    0.000    0.000    0.000 codecs.py:309(__init__)
      381    0.000    0.000    0.001    0.000 codecs.py:319(decode)
    12972    0.004    0.000    0.005    0.000 codecs.py:327(reset)
  488/122    0.000    0.000    0.005    0.000 frontendcomm.py:255(__call__)
      141    0.000    0.000    0.004    0.000 iostream.py:197(schedule)
      122    0.000    0.000    0.000    0.000 iostream.py:310(_is_master_process)
      122    0.000    0.000    0.000    0.000 iostream.py:323(_schedule_flush)
      122    0.000    0.000    0.004    0.000 iostream.py:386(write)
      141    0.000    0.000    0.000    0.000 iostream.py:93(_event_pipe)
      141    0.003    0.000    0.003    0.000 socket.py:432(send)
      141    0.000    0.000    0.000    0.000 threading.py:1017(_wait_for_tstate_lock)
      141    0.000    0.000    0.000    0.000 threading.py:1071(is_alive)
      141    0.000    0.000    0.000    0.000 threading.py:513(is_set)
 12608784    3.737    0.000    3.737    0.000 wordle.py:100(is_latest_comb)
 12556896   11.626    0.000   11.626    0.000 wordle.py:106(next_comb)
122249730   23.421    0.000  103.438    0.000 wordle.py:117(eqScore)
122249730  463.222    0.000  541.458    0.000 wordle.py:121(score)
  1999251   41.824    0.000  688.707    0.000 wordle.py:149(filter_combinations)
    51888    0.529    0.000  710.142    0.014 wordle.py:160(worst_elimination)
        4    0.342    0.086  710.488  177.622 wordle.py:171(best_guess_and_elimination)
        4    0.000    0.000  710.569  177.642 wordle.py:200(play_round)
        1    0.000    0.000  710.569  710.569 wordle.py:209(play_game)
        1    0.000    0.000  710.812  710.812 wordle.py:238(play_all_games)
        1    0.116    0.116    0.239    0.239 wordle.py:260(readFile)
        1    0.003    0.003    0.243    0.243 wordle.py:282(all_word_combinations)
      135    0.000    0.000    0.000    0.000 wordle.py:33(my_char)
    64865    0.041    0.000    0.041    0.000 wordle.py:4(my_ord)
        1    0.000    0.000    0.000    0.000 wordle.py:63(strScore)
       27    0.000    0.000    0.000    0.000 wordle.py:69(strPegs)
125571267   66.473    0.000   82.004    0.000 wordle.py:75(eqPegs)
    51888    0.020    0.000   20.985    0.000 wordle.py:86(all_scores)
    51888    4.736    0.000   20.965    0.000 wordle.py:89(all_combinations)
      381    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
        2    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
        1    0.000    0.000  710.813  710.813 {built-in method builtins.exec}
      122    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
251435012   15.549    0.000   15.549    0.000 {built-in method builtins.len}
       61    0.000    0.000    0.005    0.000 {built-in method builtins.print}
        2    0.002    0.001    0.002    0.001 {built-in method io.open}
      122    0.000    0.000    0.000    0.000 {built-in method posix.getpid}
      141    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.lock' objects}
      141    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
1235379000   79.103    0.000   79.103    0.000 {method 'append' of 'list' objects}
        2    0.000    0.000    0.000    0.000 {method 'close' of '_io.TextIOWrapper' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {method 'discard' of 'set' objects}
    12972    0.001    0.000    0.001    0.000 {method 'isalpha' of 'str' objects}
      135    0.000    0.000    0.000    0.000 {method 'keys' of 'dict' objects}
        1    0.024    0.024    0.025    0.025 {method 'readlines' of '_io._IOBase' objects}
   279498    0.024    0.000    0.024    0.000 {method 'strip' of 'str' objects}
    25944    0.003    0.000    0.003    0.000 {method 'upper' of 'str' objects}
    12972    0.005    0.000    0.009    0.000 {method 'write' of '_io.TextIOWrapper' objects}
```

Figure 3. Profiling.

This may seem surprising as the score calculation is relatively simple. Notice however:

- The code was very generic (parametric in number of holes and colors) which also made it slower.
- The calculation is tricky, since a letter in the guess can score against only a single letter in the secret.

As a simple optimization, the score of any guess against any secret was precomputed and stored in a file, which was then loaded as the first step of the algorithm. The loading took around 2 minutes, but after this the running of a single game took only 1 minute, i.e., a reduction of a factor 25. The total time for all games was down to 9 days.

At this point we arranged a weekend session with a number of co-workers, where the set of games was divided into sections and different co-worker would run the algorithm on an allocated section. The calculations in Table 2 were completed over the weekend.

numPy
The version run during the session mentioned above, has a memory consumption of around 11,5 Gb to store the score array with 168.272.784 entries. It was suggested by Jonatan Kutchinsky to use numPy to reduce the memory footprint, which would allow running multiple instances without exceeding available memory. This decreased memory usage to 3.5 Gb, and increased the run time with a factor 1.5, but this price is worth paying to be able to run parallel instance. Doing this, Jonatan covered a large fraction of the games during the session.

Running the code on the author's laptop, a MacBook Pro with 16 virtual cores and 16 Gb memory, the numPy version allowed 3 parallel instances, taking into account that some memory is needed for basic operations.

Hence this was a factor 2 reduction in run time compared to the single thread Caching Scores version.

Real Scores
In an attempt to make the algorithm more accurate (and perhaps reduce the required number of guesses needed), we changed a subtle detail in the Next Guess algorithm.

The algorithm states that, for each remaining word and each score, we should compute the number of remaining possible secrets with that score. Initially the algorithm iterated, for each remaining word, over all scores in general, and computed the number of possible secrets with that score by iterating over all possible secrets. Instead, it was changed to compute a map from scores to multiplicity by iterating over the possible secrets directly.

This gave a factor 18 reduction in run time when comparing single threaded numPy version.

C++
As the last step, the code was rewritten by Ivan Olkhovskii to C++ with efficient handling of fixed size arrays and implementation of internal threads with shared memory for the scores array. (In contrast, the Python program was just run as multiple instances from the terminal, each with a separate cache in memory.)

The resulting version was a factor 9.8 faster single threaded compared to the single threaded Real Scores version. Memory consumption was 0,5 Gb. (In reality, it was earlier versions that were translated to efficient C++ and then the subsequent improvements to the Python version were carried over to C++ by the author.)

C++ multi-threaded
Due to the lower memory consumption in general, and because the Score cache was shared between all threads, it was possible to utilize all 16 threads running in parallel. The full list of games could then be completed in 15 minutes, i.e., a factor 24 compared to the fastest Python version running with three threads. The shared memory was probably not essential, since the memory usage was sufficiently low in any case.

Cache the guesses
After completion of the first version of the paper, the author realized that yet another significant improvement would be possible.

When we compute the next guess to make, it depends only on the guesses we have made and the scores we got. In other words, if we make in a later game the same guesses and get the same scores, we will have the same next guess as in the former game.

It may seem that this reuse is unlikely to occur, as we will get new scores in new games, since we are playing against new secrets. However, notice that there are in any step at most 243 possible scores. In particular, when we make the first guess in each of the 12972 games, there are only 243 different scores, so when we make the second guess, in 12729 out of the total 12972 cases, we can get the answer from the cache.

It may seem that the gain is marginal since it is mainly relevant for the second guess (which is actually the first guess that gets computed as the first guess is fixed). However, notice that the second guess is by far the most expensive one to compute due to the large number of remaining possible secrets remaining after the first guess.

The change was tried out in the single threaded Python version and yielded a reduction in run time of a factor 6.

The cache was first implemented as an array of games, where a lookup for a new partial game simply iterated over all the games of the cache and tried to obtain a match. Subsequently, the cache was refactored to be a recursive type:

Cache: Guess x Map
Map: Score -> Cache

The difference to the array implementation was a 10% reduction in runtime on a small number of games, but it was feared that the difference would be bigger on the full set of games, so the more efficient version was adopted.

The difference between the array version and the recursive type is not indicated in Table 4. The times are for the recursive type.

The optimization was carried over to the C++ version. After some difficulties getting the recursive structure accepted by the type system (via the use of pointers), the first version was tested. Since the guess cache is a real shared cache, potentially updated by all games (in contrast to the score cache which is precomputed and then read-only), many threads failed with segmentation fault, due to the lack of handling of concurrency.

The multi-threaded C++ version ran by executing each game as a separate thread with access to the shared score cache. This did work with the computed guess cache. In order to avoid synchronization, which might affect run time negatively, the C++ program was refactored to run in a fixed number of threads (16) each with their own cache. Memory consumption was then multiplied by a factor 16, but the segmentation faults were avoided. Running all games spread over the 16 threads, took 2.5 minutes, which is a factor 7.5 reduction of the 15 minutes of the previous multi-threaded C++ version.

The pre-computation of the scores took 4 minutes, which is slower by a factor 2 than the previous version. Notice however, that if we want to run the games with different start words, the score cache does not have to be recomputed. So we disregard this slow-down.

Conclusion
The optimizations yielded a factor 567.600 improvement from the initial Python version to the final multi-threaded C++ version with caching of guesses.

The main vehicles in the optimization were the following:

1. **Precomputation** of scores, these are the same for every game and startword.
2. **Caching** of games, these vary with the start word.
3. **Parallelization** to multiple threads and CPUs.
4. **Modifying the algorithm** to avoid unneeded computations.
5. **Changing data structure** to faster operations.
6. **Changing language** to compiled code and data structures closer to physical structures.

The main parameter for optimization was running time, but reducing memory consumption also became important to facilitate precomputation and scaling to more threads.

EPILOGUE

To prepare for releasing the Python code to github [9], we made a final round of optimizations of the code.

One idea was to reduce the memory footprint (thereby enabling more parallel threads) of the precomputed scores. The point was to only compute half the scores, taking into account that—like in Master Mind—the score of a guess against a secret, is the same as the score of the secret against the guess.

However, a little reflection shows that this does not work in Wordle. For instance

- AAHED against AARGH scores 22100
- AARGH against AAHED scores 22001

The reason is that position matters in scores in Wordle.

Instead, we optimized the storage representation of the scores, by seeing scores as 2-byte integers instead of 5-character strings (or 4-byte integers plus overhead). This reduced memory for a single thread from 3.5Gb to 0.5Gb.

We also replaced manual array membership (looping through looking for an element) with the **in** operator.

Another source of optimization, is to notice that the filtering after obtaining a score, may be cached in several ways. One possibility is to notice that as we compute the min-max value for a guess, we actually compute the number of remaining possible guesses for each score. This could be changed to compute not just the number, but the actual remaining guesses and store them in a map from guess and score to the remaining possible guesses (taking into account the partial game already played). Thus, after getting the score for what turned out to be the best guess, we can look up the remaining possible guesses.

However, the necessary computations (many of which would never be used) turned out to increase overall time significantly. Instead, we chose just to cache the actual filtering done after each score is known. This means that for any score, where we get the new guess from the cache, we also get the remaining possible guesses after the score from the cache.

After these optimizations the running time for a single thread went down to around 5 minutes, i.e., around 0,023s per game, which is close to the multi-threaded C++ version.

Running 8 parallel threads with the resulting program, the process completed in around 7 minutes (as always, ignoring precomputation), amounting to 0,004s per game. This is twice as fast as the corresponding C++ code and scaling and an improvement factor of more than 1 mio from the original unoptimized single thread Python code.

However, there is a bit of cheating here compared to the C++ code, as this last Python example did not run different sets of games for the same secret, but rather all games for different secrets (which in the end, is the most relevant approach). Thus, each thread had better chance to utilize the score cache. Probably the C++ code could run/scale faster with the same modified approach.

REFERENCES

[1] *https://en.wikipedia.org/wiki/Wordle*.

[2] Collins official Scrabble Words 2019. https://boardgames.stackexchange.com/questions/38366/latest-collins-scrabble-words-list-in-text-file

[3] Kooi, Barteld. (2005). Yet another Mastermind strategy. ICGA Journal. 28. 10.3233/ICG-2005-28105. https://www.researchgate.net/publication/30485793_Yet_another_Mastermind_strategy

[4] Irving, R. (1978-1979). Towards an Optimum Mastermind Strategy. Journal of Recreational Mathematics, Vol. 11, No. 2, pp. 81–87.

[5] Koyoma, K. and Lai, T. (1993). An Optimal Mastermind Strategy. Journal of Recreational Mathematics, Vol. 25, No. 4, pp. 251–256.

[6] Knuth, D. (1976-1977). The Computer as a MasterMind. J. Recreational Mathematics, Vol 9(1). https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

[7] https://supermastermind.github.io/playonline/optimal_strategy.html

[8] https://puzzling.stackexchange.com/questions/114316/whats-the-optimal-strategy-for-wordle

[9] https://github.com/morten-heine/wordle